Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Tanulási keretrendszer a C++ programozáshoz

Témavezető:

**Pataki Norbert**

Adjunktus, PhD

Szerző:

**Török Richárd Dávid**

Programtervező informatikus, Bsc

Budapest, 2021

Tartalom

[1 Bevezetés 3](#_Toc60141662)

[1.1 Témaválasztás indoka 3](#_Toc60141663)

[1.2 Megoldandó feladat 3](#_Toc60141664)

[2 Felhasználói dokumentáció 5](#_Toc60141665)

[2.1 Rendszerkövetelmények 5](#_Toc60141666)

[2.2 Telepítés 6](#_Toc60141667)

[2.2.1 Első indítás 6](#_Toc60141668)

[2.2.2 Már inicializált adatbázissal 8](#_Toc60141669)

[2.3 Content management system felhasználói felület 8](#_Toc60141670)

[2.3.1 Bejelentkezési képernyő 8](#_Toc60141671)

[2.3.2 Kezdő képernyő 9](#_Toc60141672)

[2.3.3 Leckék szerkesztése 10](#_Toc60141673)

[2.3.4 Feladatok szerkesztése 12](#_Toc60141674)

[2.3.5 Nyitó oldal szerkesztése 13](#_Toc60141675)

[2.4 Alkalmazás felhasználói felület 14](#_Toc60141676)

[2.4.1 Kezdő képernyő 14](#_Toc60141677)

[2.4.2 Lecke nézet 15](#_Toc60141678)

[3 Fejlesztői dokumentáció 19](#_Toc60141679)

[3.1 Alkalmazás felépítése 19](#_Toc60141680)

[3.2 Statikus megjelenési tartalom 20](#_Toc60141681)

[3.2.1 Könyvtárszerkezet, komponensek 21](#_Toc60141682)

[3.2.2 Környezeti változók 22](#_Toc60141683)

[3.2.3 Komponensek hierarchiája 24](#_Toc60141684)

[3.3 Kódfordítást végző szerver 26](#_Toc60141685)

[3.3.1 Architektúra 26](#_Toc60141686)

[3.3.2 API 29](#_Toc60141687)

[3.3.3 Build 33](#_Toc60141688)

[3.3.4 Környezeti változók 33](#_Toc60141689)

[3.4 Adatbázis 34](#_Toc60141690)

[3.4.1 Táblák 34](#_Toc60141691)

[3.4.2 Környezeti változók 37](#_Toc60141692)

[3.4.3 Az adatbázis elérése 37](#_Toc60141693)

[3.5 Tesztelés 40](#_Toc60141694)

[3.5.1 Kódfordító szerver tesztelése 40](#_Toc60141695)

[3.5.2 ReactJS alkalmazás tesztelése 42](#_Toc60141696)

[4 Összefoglalás 47](#_Toc60141697)

[4.1 Továbbfejlesztés lehetőségei 47](#_Toc60141698)

# Bevezetés

## Témaválasztás indoka

A koronavírus által bekövetkezett helyzet rámutatott, hogy milyen nagy mértékű szükségünk van az online felületen is elérhető tananyagokra. Ebből az okból írtam, mint szakdolgozatomként, egy online felületen elérhető oktató jellegű alkalmazást. A szakdolgozatom célja, hogy egy olyan környezetet biztosítson a C++ programozási nyelvnek, hogy azt könnyen fogyasztható formában lehessen elsajátítani.

Az alkalmazásban a C++ programozási nyelv ismereteit leckék által lehet elsajátítani. A leckékhez feladatok is tartoznak, így a felhasználó egyből tudja gyakorolni a frissen szerzett tudását, amellyel sokkal jobban rögzíteni is tudja azt a már elsajátított tudást az emlékezetében. A leckék különböző nehézségi szintek szerint vannak beosztva, így a felhasználó a saját képességeihez mérten választhat a tananyagokból, és fokozatosan fejlesztheti tudását. A leckék egy diasorszerű felületen vannak prezentálva, amik között a felhasználó a navigációs gombok segítségével lépegethet az anyagban.

Mind amiatt, hogy a tudás jobban rögzülhessen a tanuló személyében, az elméleti anyagok mellett gyakorlatban is kipróbálhatja, gyakorolhatja a már megszerzett tudását. Az alkalmazás ugyanis tartalmaz egy online fordítót is, amivel a lecke tanulása alatt kipróbálható az újonnan megszerzett ismeret, illetve a programozási feladatokban próbára tehető az elsajátított tudás is. Ennek működéséről a dokumentáció későbbi szakaszában lesz szó.

## Megoldandó feladat

A projekt célja, hogy a végfelhasználónak ne kelljen semmilyen extra programot telepítenie, ezáltal kényelmesebben lehessen használni, és sokak által hozzáférővé váljon. Egy modern böngészőre legyen szüksége mindösszesen, hogy elérhető legyen számára az oktató felület. A kódfordításhoz nem kell, hogy C++ fordítóprogram legyen az eszközünkön, ez mind szerveroldalon történik.

A kód fordítása és futtatása egy külön Docker konténerben történik, amely szabályozható életidővel rendelkezik. Mindez biztonság, pontosabban elővigyázatosság miatt, ugyanis ezáltal ki tudjuk küszöbölni, hogy esetlegesen egy káros kód fusson le a szerveren, amely kártékony hatásokat hajthat végre a szerver rendszerében.

Az oldal irodalmi tartalmát egy headless CMS[[1]](#footnote-1) szolgáltatja. Ennek tartalma dinamikusan változtatható egy külön felhasználói felületen, ahol akár több szerkesztő felhasználót is létrehozhatunk. Ezen a felületen tudunk új leckéket és feladatokat is létrehozni. Ez a rendszer tartalmaz egy publikációs megoldást is, ami segítséget nyújt, hogy csak az a tartalom jelenjen meg az alkalmazásban, aminek már végeztünk a szerkesztésével.

A projekt készítése során próbáltam egy olyan tanulási módszertant követni, miszerint a felhasználó nem csak megfigyelő a lecke során, hanem aktívan ki is próbálhatja az új ismereteket, és a saját tempójában haladhat az anyaggal.

# Felhasználói dokumentáció

A teljes alkalmazás dockerizálva van, ha lokálisan szeretnénk futtatni, nincs szükségünk NodeJS-re, vagy egyéb rendszerekre. Két alkalmazást kell telepítenünk, a [Docker Desktop](https://www.docker.com/products/docker-desktop)-ot, illetve a [Postman](https://www.postman.com/downloads/)-t. A Docker biztosítja már azt a lehetőséget is, hogy a virtualizációhoz Windows-os konténereket használhassunk, viszont ennek ellenére sokkal jobban ajánlott a Linux alapú konténereket választani, ugyanis ilyen alappal lett maga az alkalmazás is fejlesztve, ezen okból nem várt problémák adódhatnak ellenkező esetben. Pontosabban akkor, amikor is a nem Linux alapú konténereket használjuk. A Postman az adatbázis inicializálásának megkönnyítéséhez szükséges.

## Rendszerkövetelmények

Docker Desktop for Windows esetén:

* Windows 10 (64-bit) Pro, Enterprise, vagy Education:
  + Build 16299 vagy nagyobb
  + Hyper-V és a Containers Windows funkció engedélyezve kell, hogy legyen.
  + Hardveres követelmények:
    - 64 bites processzor SLAT támogatással
    - 4GB memória
    - BIOS-szintű hardveres virtualizáció engedélyezése, [bővebben](https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled)
* Windows 10 (64-bit) Home:
  + 1903-as vagy nagyobb verzió
  + WSL 2 funkciók engedélyezése, [dokumentáció](https://docs.microsoft.com/en-us/windows/wsl/install-win10)
  + Hardveres követelmények:
    - 64 bites processzor SLAT támogatással
    - 4GB memória
    - BIOS-szintű hardveres virtualizáció engedélyezése, [bővebben](https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled)

Illetve egy modern böngésző, az alábbi verzió számokkal vagy nagyobbal:

* Chrome (86.0.4240.198)
* Edge (87.0.664.55)
* Firefox (82.0.2)

## Telepítés

Miután sikeresen feltelepítettük a Docker Desktop és Postman alkalmazást, a projektet a következőképpen tudjuk elindítani.

### Első indítás

Az első indítás kicsit bonyolultabban végezhető el, mint az elkövetkezendő többi indítás, ugyanis inicializálnunk kell az adatbázist. Az adatbázis volume-ok segítségével van fenntartva futások között, így élettartama a konténerekétől független. Mindebből az következik, hogy a későbbi indításokkor már nem kell az összes következő lépést végrehajtanunk.

Lépések:

* Service-ek elindítása és adatbázis inicializálása
  + *docker-compose up --detach --scale compiler=0*
  + *docker-compose run --rm --entrypoint "/bin/sh -c" directus "/var/directus/bin/directus install:database"*
* Admin felhasználó létrehozása
  + *docker-compose run --rm --entrypoint "/bin/sh -c" directus "/var/directus/bin/directus install:install -e <email> -p <password> -t C++ -T <access-token>”*
  + Itt az <email> és <password> helyére tetszőleges értékeket írhatunk be.
  + Az <access-token> helyére diploma-2020 értéket írjunk, [lentebb](#access_token) olvasható mi a teendő, ha más értéket szeretnénk

Példa:

2‑1. ábra sikeres indítás konzol kimenetele

Adatbázis feltöltése

* + Nyissuk meg a Postman alkalmazást
  + Importáljuk be a /cms/postman\_collection fájlt, majd futtassuk le

2‑2. ábra collection futtatása a postman alkalmazásban

* + Állítsunk be 1 másodperces késést a kérések között. A 2‑3. ábra bemutatja a sikeres lefutás eredményét.

2‑3. ábra Collection futtatásának eredménye a postman alkalmazásban

* Ha ezeket a lépéseket elvégeztük az, alkalmazás futásra kész állapotba került.

Saját access-token érték esetén:

A ’diploma-2020’ értéktől eltérő access token esetén a következő lépéseket kell megtennünk:

* A /frontend/.env fájlban a **REACT\_APP\_ACCESS\_TOKEN** kulcs értékének a telepítéskor megadott <access-token> értéket kell beállítani.
* Újrabuildelni a frontend service-t a *docker-compose build frontend* paranccsal
* Leállítani a jelenlegi serviceket (*docker-compose down*)
* Majd újraindítani (*docker-compose up --detach --scale compiler=0*)

### Már inicializált adatbázissal

Ha már korábban elvégeztük 2.2.1 lépéseit, a következőképpen tudjuk elindítani az alkalmazást:

* *docker-compose up --detach --scale compiler=0*

## Content management system felhasználói felület

Az alkalmazásban megjelenő tartalmat ezen a felületen keresztül tudjuk szerkeszteni, illetve bizonyos entitások státuszát változtatni.

A felületet az alapbeállításokkal a szerver 8080-as portján tudjuk elérni. Esetünkben ez a lokális szerver, tehát az elérési útvonal a következő: <http://localhost:8080>.

A port számot módosíthatjuk, ha a projekt gyökér szintjén levő docker-compose.yml fájlban átírjuk a 2‑4. ábra által mutatott értéket.



2‑. ábra Részlet a docker-compose.yml fájlból

### Bejelentkezési képernyő

Meglátogatva a fentebb található linket, a 2‑5. ábra által mutatott felülettel találkozhatunk. Itt a telepítésnél megadott email és jelszó párossal léphetünk be. A rendszer sütik segítségével megjegyzi a belépési adatainkat, így azokat nem kell minden alkalommal újra megadnunk.



2‑. ábra CMS belépő képernyő

### Kezdő képernyő



2‑. ábra CMS főmenü

A 2‑6. ábra bemutatja kékkel jelölt dobozban a collection-öket. Ezek az egyes elemek csoportosítására szolgálnak. Itt tudjuk az elemeket kilistázni, bizonyos elemekre szűrni, illetve új elemet hozzáadni. Fontos megjegyezni, hogy a Home collection egyke, ami annyit jelent, hogy a másik kettővel ellenkezőleg csak egy darab példány létezhet. Rákattintva egyből annak az egy elemnek a szerkesztő képernyője jelenik meg.

A 2‑6. ábra mutatja zölddel jelölt résszel a felhasználók kezelésére szolgáló felület. Itt lehet új felhasználót hozzáadni, vagy meglévő adatait, vagy státuszát szerkeszteni.

Piros színnel pedig a cms-be feltöltött fájlokat-, média tartalmakat tekinthetjük meg, illetve adhatunk hozzá újakat.

### Leckék szerkesztése



2‑. ábra CMS-ben kilistázott leckék

Kétféleképpen tudjuk megjeleníteni a részletes szerkesztői nézetet. A 2‑7. ábra által mutatott kék színnel jelölt hozzáadás gomb megnyomásával, ekkor egy új leckét hozunk létre. Illetve, ha a kilistázott elemek egyikére rákattintunk. A státusz oszlopban a kék pötty jelzi azt, hogy az adott lecke publikálva van, és listázva van az alkalmazásban. Piszkozat esetén ez szürkével jelölt.

Egy leckének nyolc szerkeszthető mezője van:

* status
* name
* url\_alias
* difficulty
* icon
* description
* slides
* exercise

Status:

Három választási lehetőségünk van, *Published, Draft, Deleted*. Draft esetén nincs listázva az adott lecke az alkalmazásban. Fontos, hogy csak olyan leckét állítsunk Published-re, amihez a feladatok részét is elkészítettük már, és össze is kötöttük a kettőt az exercise mező segítségével a lecke szerkesztői képernyőn. Ez egy lenyíló mező, ahol kiválaszthatjuk név alapján a megfelelő feladatot.

Url\_alias:

Ez egy kötelező szövegmező. Az angol ábécé kis betűi, illetve kötőjel használható benne. Ez a szöveg fog megjelenni a böngésző url részében, ha megnyitjuk a leckét.  
Példa:



‑. ábra url alias megjelenése

Difficulty:

Ez egy kötelező lenyíló mező, ahol a lecke nehézségi szintjét adhatjuk meg. Három lehetőségünk van: *Easy, Intermediate, Professional*. Azt befolyásolja ez a mező, hogy a menüben melyik kategória alá kerüljön a lecke, illetve jelzést is ad a felhasználónak a lecke nehézségéről.

Icon:

Opcionális mező, a leckekártyán megjelenő illusztráció. Ha nem adjuk meg, helyét a nehézséghez rendelt szín tölti ki.

2‑9. ábra lecke kártyák

Description:

Kötelező szövegmező, ami összefoglalja a lecke lényegét, és érdeklődést kelt fel. A 2‑9. ábra bemutatja ezeket a leckekártyákat, amin megjelennek. Illetve a megnyitott lecke neve alatt is látható.



2‑10. ábra lecke leírás és cím

Slides:

Ez egy összetett mező, itt lehet megadni a tartalom lényegi részét. A lecke anyaga egy slideshow szerű felületet van prezentálva. Két entitásunk van, slide és step. Egy slide-on belül több step-ünk is lehet, és egy step-ben lehet kódrészlet vagy szöveges tartalom. A slideshow-n van egy léptető, amivel a következő step-et jeleníthetjük meg, vagy visszaléphetünk az előzőre. Ha el szeretnénk tüntetni az előző step-eket a slideshow-ról, kezdjünk egy új slide-ot és arra rakjunk új step-eket. A szöveg valójában HTML tartalmat hoz létre, tehát be lehet szúrni képeket és multimédiás tartalmat is, továbbá különböző méretű címsorokat. Van egy teljes képernyős gomb is, amivel megkönnyebbíthetjük a szerkesztési élményt. Ezt a 2‑11. ábra ábrázolja, a világosabb szürke hátterű gombbal. Fontos, hogy igaz két mezőt látunk egy step-en belül, de csak az egyikbe írjuk tartalmat, így elkerülve a sorrendiségi problémákat. A step-eket, illetve a slide-okat átrendezhetjük, ha áthúzzuk a sorrendjüket a baloldali sáv segítségével.

2‑11. ábra slide szerkesztése

Exercise:

Lenyíló mező, a rendszerben tárolt leckéket listázza ki név szerint.

### Feladatok szerkesztése

Meglévő feladatot ugyanúgy szerkeszthetünk vagy hozhatunk létre, mint leckék esetén. Egy leckéhez tartozó feladat kétfajta feladatból állhat.

**Kvíz feladatok**, ezeknek száma 0-tól a kívánt mennyiségig terjedhet. A kvízeket a 2‑12. ábra által mutatott kvíz szerkesztő felületen lehet megadni. Nem csak egy helyes válasz lehetséges. Egy kis kapcsolóval lehet állítani, hogy a választási lehetőség helyes-e. Sajnos a CMS rendszerben van egy ismert hiba, nem menti el a kezdetben hamisnak jelölt és úgy is maradt opciókat. Ennek egy egyszerű, de kissé idegesítő megoldása, ha egyszerűen igazra, majd hamisra állítjuk a kis kapcsolót.

****

2‑. ábra Kvíz szerkesztése

**Programozási feladat,** ebből feladatsoronként egy létezhet. Ennek az entitásnak három mezője van, *question, description, solution*. A description kivételével a többi kötelező mező. Az opcionális description mezőbe írhatunk tippeket a feladat megoldásához, vagy feladat specifikációt. Fontos kihangsúlyozni, hogy a rendszer úgy ellenőrzi a feladatot, hogy a program output-ját hasonlítja össze a solution mezőbe írt értékkel. Itt a betűméret nem számít, a program átalakítja a program kimenetét és a megoldás szövegét egyaránt kisbetűssé.

### Nyitó oldal szerkesztése

A home collection alatt tudjuk szerkeszteni a kezdőképernyőn megjelenő tartalmat, a láblécben megjelenő kontakt emailt, és telefonszámot, valamint az oldalsó menüben megjelenő ’hasznos linkek’ szekciót. Ha nem adunk meg egyetlen linket sem, akkor maga a szekció címkéje se fog megjeleni az oldalsó menüben. Maga a kezdőoldalon megjelenő tartalom a *content* mezővel szerkeszthető, ez is egy HTML tartalmat generál, el lehet helyezni benne különböző méretű címsorokat, multimédiás tartalmakat, képeket, és még sok mást.

## Alkalmazás felhasználói felület

### Kezdő képernyő



‑. ábra kezdő képernyő

Az alkalmazást betöltve az 2-13. ábrához hasonló nézettel találkozunk. Itt a tartalom a szerkesztőtől függ. Ezen az oldalon különböző cikkeket olvashatunk, amit a szerkesztő létrehoz, vagy esetleg más hivatkozásokat is hozzáadhat. Alul a láblécben található a kontakt email és telefonszám. Ezek linként is szolgálnak, rájuk kattintva egyből elindul a számítógépünkön alapértelmezett levelező, vagy telefonhívást végző alkalmazás. A bal felső sarokban található a menü gomb. A 2‑14. ábra által bemutatott nézetet kapjuk a menüt megnyitva.



2‑. ábra kinyitott menü

A C++ ikonra kattintva visszajutunk a főoldalra. Az útmutatók címke alatti könnyű, közepes és haladó menüpontok valójában összecsukható almenük, amik újabb menüpontokat nyitnak meg. Bármely kategóriának az ’összes’ menüelemére kattintva a 2‑15. ábra által bemutatott összegző oldalra jutunk. A leckekártyákra kattintva a választott tananyaghoz navigál az applikáció.



2‑. ábra összegző képernyő

### Lecke nézet

A 2‑16. ábra mutatja a lecke nézet. Itt a főoldalhoz hasonlóan megmarad az oldal szerkezete, fejléc benne a menüvel és a lábléc. Az oldal tartalma változik csak.



2‑. ábra lecke nézet

Az első sorban a lecke címe található, alatta egy menü, amivel válthatunk a lecke és feladatok nézet között. A menü alatt pedig a lecke leírása található. Itt az egész szöveg megjelenik, mivel előfordulhat, hogy a leckekártyán a szöveget levágjuk, ha az túl hosszú. A leírás alatt pedig a diasorok találhatóak, amiben a lecke anyaga van. Itt a nyilakkal jelölt gombokkal lehet előre vagy hátra haladni az anyagban.



‑. ábra diasor

Alul egy rózsaszín csík jelzi hol tartunk éppen az anyagban. Ha az anyag végére értünk, a 2‑18. ábra által bemutatott képernyő jelenik meg.



2‑. ábra befejezett lecke

Három opciót kínál fel nekünk a felület. Újrakezdhetjük a tananyagot az elejétől, a feladatok szekcióra ugorhatunk, vagy visszamehetünk a jelenlegi lecke szintjével megegyező összegző képernyőre.

A diasor alatt található egy online kódfordító rész, ezt a 2‑19. ábra illusztrálja. Ez három fülből áll, code, arguments és output. A code fülön tudjuk szerkeszteni a forráskódot, amit majd le szeretnénk futtatni, az arguments fül alatt adhatjuk meg az argumentumokat, amiket a program kap meg futtatáskor. Az output fülön pedig a program futásának kimenetelét láthatjuk. Ha hibás a forráskódunk és fordítási hibánk van, azt is ezen a fülön láthatjuk. A fordít gomb lenyomásával indíthatjuk el a fordítási folyamatot, ameddig ez fut a gomb inaktív állapotba kerül. Ha végzett a folyamat átvált automatikusan az output fülre.

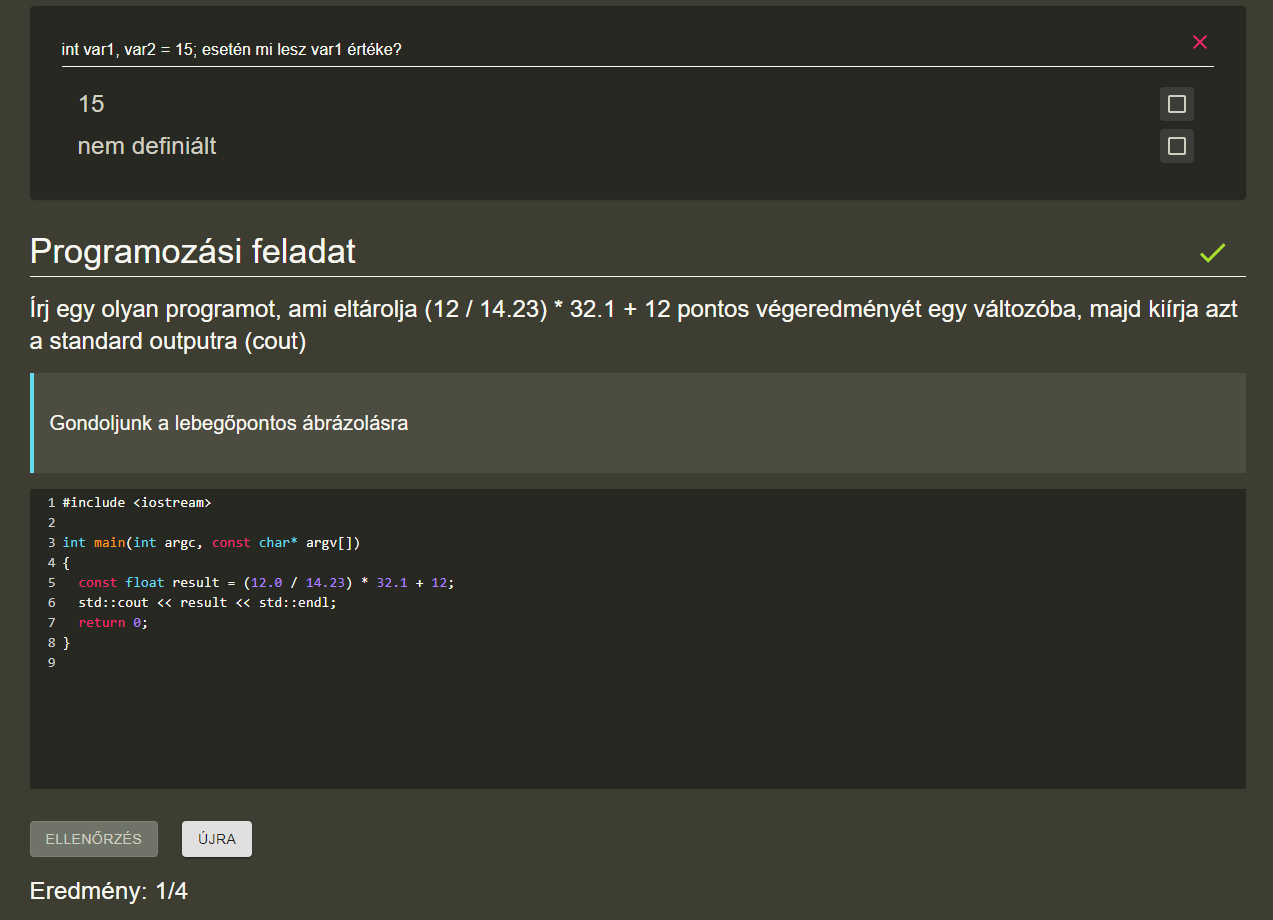
Három lehetséges végződése van a folyamatnak:

* Sikeresen lefut a program, ekkor a program kimenetele kerül kiírásra
* Fordítási hibánk van, ekkor a fordító által írt hibaüzenetet írjuk ki
* A folyamat túllépi az időkorlátot, ekkor a *timed out* üzenetet írjuk ki



2‑. ábra online fordító

A feladatokat nézetre kétféle módon tudjuk eljutni, ha a fenti almenüben a ’Feladatok’ menüpontot választjuk (2‑16. ábra), vagy miután végeztünk a leckével a középső ’A feladatokhoz’ opcióra kattintunk. Először a kvízek vannak kilistázva, ha az adott leckéhez tartozik legalább egy darab, utána pedig a programozási feladat.



2‑. ábra Leellenőrzött feladatmegoldás

Az ellenőrzés gombra kattintva, elküldjük a válaszainkat kiértékelésre. Ameddig ez folyamatban van az ’újra’ és ’ellenőrzés’ gomb is inaktív állapotba kerül. Ha végzett a folyamat a kiértékeléssel minden kérdést megjelöl egy zöld pipával vagy piros X-szel a válasz helyességétől függően. Erre a 2‑20. ábra ad példát. A gombok alatt pedig kiírja mennyi kérdésre adtunk jó választ az összesből. Az újra gomb megnyomásával a feladatlap visszaáll az eredeti állapotába és újból próbálkozhatunk. Célszerű megjegyezni, ha feladatok és lecke fül között váltakozunk, a feladatlap mindig visszaáll a kezdetleges állapotába, így nem lehet olyan könnyen lenézni a választ a diasorról.

# Fejlesztői dokumentáció

## Alkalmazás felépítése

Az alkalmazás minden modulja egy Docker konténerben van elhelyezve, így könnyítve a jövőbeli a skálázhatóságot, és az éles környezetbe való helyezést. A forráskódban több docker-compose fájl is szerepel, ha esetleg a teljes alkalmazás egy kisebb részét szeretnénk lebuildelni, vagy futtatni. A projekt gyökér szintjén található a fő compose fájl. Az alkalmazásban szereplő összes modul/service megtalálható benne.

A compose fájlok fontosságát érdemes kiemelni, ugyanis ezek sok kényelmi funkcióval szolgálnak, mint például volume-ok helybeli létrehozása, amivel a futások közti adatokat tudjuk fenntartani, vagy például egy alap hálózat, amin keresztül a konténerek tudnak kommunikálni, és egyfajta címfordítást is biztosít, hogy könnyebben tudjuk megadni az elérési címeket más konténereknek. Emellett maga a konténerek példányosítását és konfigurálását is megkönnyíti.

Az alkalmazás architektúrája a következőképpen ábrázolható:



ábra ‑ az alkalmazás szerkezete

14.15.0-as NodeJS verziót használtam az ezen futó részalkalmazásokhoz. A build folyamat- és a futtatás során használ verzió megegyezik. A build egy ideiglenes Docker konténerben készül, nem szükséges, hogy saját eszközünk rendelkezzünk ezzel a pontos verziószámmal. Ezeknek a konténereknek 3.12-es verziójú alpine linux az operációs rendszere. Ez biztosítja a lehető legkisebb végleges image méretet.

A Docker image egy modellszerű fájl, aminek segítségével az image-ben meghatározott specifikációjú konténereket tudunk létrehozni.

## Statikus megjelenési tartalom

A frontend elkészítéséhez a ReactJS keretrendszert használtam. A [create-react-app](https://www.npmjs.com/package/create-react-app) script által generált sablon projektet vettem alapul. Ez magába foglalja a Jest unit tesztelési keretrendszert, a Webpack bundle toolt, ami a build elkészítésében segít, illetve egy fejlesztői szervert is tartalmaz. Ez a *hot reload* tulajdonsága miatt is hasznos, biztosítja a forráskód újrafordítását változása esetén. Ekkor az új tartalom egyből megjelenik a böngészőben. Így elkerülhetjük, hogy mindig nekünk kelljen kézzel újraindítani a szervert. A csomag tartalmazza még a BabelJS-t, ami a transpile-olásához szükséges, hogy új még kísérleti fázisban, vagy bizonyos böngészőkben nem támogatott szintaktikát is használni tudjunk. A transpile-olás egy olyan folyamat, amikor egy bizonyos Javascript forráskódot azzal megegyező szemantikájú, de legtöbb esetben eltérő szintaktikájú kóddá alakítunk át.

A fejlesztést 16.13-as verziószámú React-al végeztem, de ennél újabb verziókkal is kompatibilis, de figyeljünk, hogy a react-scripts csomag verziószámát is frissítsük, ha újabb verziójú React-al szeretnénk a további fejlesztést végezni. Google által megalkotott Material formatervezés határozza meg a komponensek főbb stílusát. A legtöbb általános komponens ennek okán a [@material-ui](https://www.npmjs.com/package/@material-ui/core) csomagból származik. Az online forráskód szerkesztő részhez a [Codemirror](https://www.npmjs.com/package/codemirror) csomagot használtam, széles elterjedettsége, és egyszerű használata miatt. Jelenlegi állapotában a frontend projekt nem tartalmaz harmadik féltől származó csomagból eredő sebezhetőséget.

A kódformázási stílus betartását a [Prettier](https://www.npmjs.com/package/prettier) csomag segíti. Ennek konfigurációja a frontend mappa gyökerében található a .prettierrc.json fájlban. A statikus kódellenőrzést az [Eslint](https://www.npmjs.com/package/eslint) végzi. Ennek konfigurációja a .eslintrc.js fájlban található. Ez az eszköz felhívja a figyelmünket az esetleges szintaktikai hibákra, és olyan hiányosságokra, amik később szemantikai hibákhoz vezethetnek, például a függőségi tömb hiányos kitöltése React effektek esetén. Az Eslint nem végez formázást, csakis ellenőriz.

### Könyvtárszerkezet, komponensek

React v16.8-ig komponenseket, főleg csak Javascript osztályok segítségével tudtunk definiálni. Lehetett függvényekkel is, de ezek nem rendelkezhettek belső állapottal, illetve életciklus metódusokkal. Ezeket nevezzük *pure components*-nek. A 16.8-as verziótól kezdve ez viszont megváltozott a [hook](https://reactjs.org/docs/hooks-intro.html)-ok bevezetésével. Ez sok problémát kiküszöbölt, amik az osztályként definiált komponensekből eredtek. Ilyen volt például a Javascript-ben a többi nyelvtől jelentősen eltérő, és olykor kaotikusnak tűnhető *this* kulcsszó használata, sok csomagoló (wrapper) komponens és időnként nehezen értelmezhető/olvasható forráskód. Ennek elkerülésére a projektben már az újféle mintát követő functional komponenseket használom hook-okkal.

A komponens stílusok egyénire szabásához [css-in-js](https://cssinjs.org/?v=v10.5.0)-t használok. Ennek előnyei közé tartozik gyorsasága és biztonsága. Minden szabályhoz különböző azonosítót generál, így elkerülhető a névütközés, ami CSS-ben sokszor okoz fejfájást.

Könyvtárszerkezet:

 A komponensek a view könyvtárban találhatóak. Az általánosabbak, amik több helyen is fel vannak használva, a view-en belül a *components* mappában vannak.

3‑2. ábra frontend könyvtár struktúra

A bonyolultabb komponensek saját mappát kaptak, ebben általában a következők fájlok szerepelhetnek:

* Maga a komponens forrás fájlja, ennek elnevezése megegyezik a mappájáéval
* Egy styles.js fájl, amiben a stílus definíciók találhatóak. Ez egy Reack hook-ot tartalmaz. A hook-ot a [*makeStyles*](https://material-ui.com/styles/api/#makestyles-styles-options-hook)nevű függény készíti el, ami a @material-ui/styles könyvtár része. A *makeStyles* a függvénynek paraméterül egy függvényt kell megadni. Ez első paraméterként megkapja a téma objektumot, amiből kinyerhetőek a témában definiált színek és eltartások, és egy objektummal tér vissza, aminek a kulcsai lesznek a CSS osztályok. Ez a stíluslap fordításkor fog legenerálódni, és a DOM-ban a *head* részbe kerül beszúrásra. Több stíluslap esetén a beszúrások sorrendjét a *makeStyles* függvényhívások sorrendje határozza meg.
* Bonyolult állapotrendszer esetén egy state.js. Ez tartalmazza az állapotrendszer kezdeti értékét leíró objektumot, és a *reducer* függvényt. Ami azt írja le, hogy bizonyos akciók esetén, hogyan változzon az állapottér.
* A teszt fájl
* Egy index.js, ami csak újra kiexportálja a szomszédos fájlok tartalmát. Ez a [node modul rezolúciós algoritmusa](https://nodejs.org/api/modules.html) miatt érdekes, hogy a rövidebb formában (.../view/Component) lehessen hivatkozni rá, és ne kelljen a teljes elérési útvonalat megadnunk (.../view/Component/Component), ha használni szeretnénk a komponenst.

Az *api* könyvtárban két fontos fájl található:

* axios.js, ennek segítségével kommunikálhatunk a kódfordító szerver REST API-jával
* directus.js, ő pedig a CMS-sel való kommunikációt egyszerűsíti le

API hívások az alkalmazásban két esetben történnek, ha betöltjük az alkalmazást, vagy ha megnyitunk egy leckét. Ez a felhasználói élmény javítása miatt van így, hogy ne kelljen a felhasználónak sok töltőképernyőt néznie. Ennek viszont hátulütője websocket hiányában, ha új lecke kerül publikálásra amíg nyitva van az alkalmazás, erről csak akkor veszünk tudomást, ha újrafrissítjük az oldalt.

A *theme.js*-ben találhatjuk az alap material dizájnt felülíró egyedi stílusokat. Itt vannak deklarálva a téma színei is. Egy esetleges téma színvilág változtatás keretében itt a colors nevű objektum tagváltozóinak értékét kell módosítanunk. Ezen a fájlon kívül nem található direktben megadott színkód. Ha itt megváltoztatjuk az értékeket, az az egész applikációra kihatással lesz.

### Környezeti változók

Lehetőségünk van a frontend projektet környezeti változókkal konfigurálni, hogy eltérő környezetek esetén ne a forráskódot kelljen szerkeszteni. Erre a react-scripts csomag miatt van lehetőség. Fejlesztői módban az *npm start* paranccsal tudjuk elindítani az alkalmazást a frontend könyvtárból. Ekkor valójában a *react-scripts start* parancs fut le. A react-scripts sok parancsot, bonyolult konfigurációt és ezek végrehajtásához szükséges csomagokat tartalmaz, amit elrejt előlünk, hogy egy egyszerű interfészt tudjon biztosítani. Az egyik ilyen függősége a react-scripts-nek a [dotenv](https://www.npmjs.com/package/dotenv) csomag. Ez arra ad lehetőséget, hogy környezeti változókat állítsunk be egy *.env* elnevezésű fájl használatával. Ez a fájl egyszerű kulcs-érték párokat tartalmaz. A változók nevét konvenció szerint csupa nagybetűvel szoktuk írni, és szóköz helyett pedig alulvonást használunk. Minden olyan környezeti változót, amit a React alkalmazásban szeretnénk használni *REACT\_APP\_* prefixummal kell feltünteti. Például *REACT\_APP\_MY\_VAR*. Ez biztonsági célt szolgál, hogy nehogy nem oda szánt környezeti változókhoz is hozzáférhessen az applikáció. A Dotenv csomagot alap esetben lehet konfigurálni, hogy milyen elérési útvonalon találja a *.env* fájlt. Sajnos mivel én a react-scripts-et használom. Ez jelenleg még nem biztosít interfészt, hogy ezt konfigurálni lehessen. Az alapbeállítást kell használnunk, ami a projekt gyökér könyvtára. Jelen esetünkben a *frontend* mappa, nem pedig az egészet átfogó *diploma* könyvtár. Ez azzal a kellemetlenséggel jár, hogy több *.env* fájlt kell létrehozzunk, egyet külön a *frontend* mappában is. Erre megoldást szolgál a *react-scripts eject* parancsa, ami az összes konfigurációt kibontja nekünk és átadja a teljes irányítást a konfiguráció felett. Ezzel viszont vigyáznunk kell, innentől kezdve nekünk kell a különböző build tool-ok frissítését és konfigurálását elvégezni. Ez a folyamat visszafordíthatatlan, hacsak nem egy korábbi eject előtti commit-ra állunk vissza a verziókövető rendszer segítségével.

Környezeti változók listája:

* REACT\_APP\_BACKEND\_URL, a kódfordítást végző szerver elérési címe
* REACT\_APP\_ACCESS\_TOKEN, a [telepítésnél](#_2.2.1_Első_indítás) megadott access-token
* REACT\_APP\_API\_BASE\_URL, a CMS szerver elérési címe

Mind a három érték kötelező.

Fejlesztés során megváltoztathatjuk a .*env* fájl tartalmát, de ekkor újra kell indítsuk a fejlesztői szervert (npm start). A kész lebuildelt frontend alkalmazást viszont újra kell buildelni, ha azt szeretnénk hogy a *.env* fájl módosításait lekövesse.

### Komponensek hierarchiája

A komponensfa gyökere az *App.js* fájlban található. Itt van a téma stílusát-, és az applikáció főbb állapotát kiszolgáltató provider-ek. Ezáltal az összes részfában található komponens el tudja érni ezeknek az értékeit. Provider-ek segítségével egy kontextust fogyasztóinak/használóinak (Consumer-eiknek) tudunk adatot eljuttatni, prop hammering nélkül. Prop hammering-nek nevezzük azt a technikát, amikor egy komponens nem használja fel a kapott property-t, hanem csak továbbadja a gyerek komponensének.

A providereken belüli komponens szerkezetet a 3‑3. ábra írja le.



3‑3. ábra A fő komponens struktúrája

Az egész vizuális szerkezetet a *PageSkeleton* komponens öleli át. Ez gondoskodik a fejléc, lábléc és oldalsó menü elhelyezéséről. Maga a tartalom az url cím szerint ágazik szét, ezt a [react-router-dom](https://www.npmjs.com/package/react-router-dom) csomag segítségével valósítottam meg. A 3‑3. ábra ezeket az ágakat szaggatott vonallal jeleníti meg. Minden esetben egy ág van kirenderelve, az amelyik útvonalon éppen vagyunk, így javítva a memóriahasználatot. A *HomePage* komponens a ’/’ útvonalon jelenik meg, a *Tutorials* pedig a ’/tutorial’ és ennek alváltozatain, például ’/tutorial/lecke’. Ezen útvonalakon kívül bármely más a 404-es nem talált oldalra visz. Az applikáció fő kontextus értékének beállítását is ez a komponens végzi. Ameddig folynak a CMS-felé irányított kérések, az oldal tartalmi része nem kerül megjelenítésre, ilyenkor egy töltőcsíkot láthatunk a fejléc alatt.

A *HomePage* és *NotFoundPage* levélszintű komponensek, gyerekei már natív HTML elemek.

A 3‑4. ábra bemutatja a *Tutorials* komponens felépítését:



3‑. ábra Tutorials komponens struktúrája

Ez a komponens a *PageSkeleton* komponensnél is használt útvonal alapú elágazást valósítja meg. A *CategoryAll* és *All* komponensek nagyon hasonlóak, csak a kilistázott tartalomtól térnek el. A *CategoryAll* komponens kap egy *category* nevű property-t, ami alapján eldönti, hogy melyik tartalmakat kell kilistáznia. A kilistázáshoz szükséges adatokat a gyökér szinten található *AppProvider*-ben létrehozott kontextusból nyeri. A *Tutorial* komponens a CMS-ben megadott *url\_alias* segítségével tudja lekérni a neki szükséges adatokat.

A *Tutorial* komponens:



‑. ábra Tutorial komponens felépítése

Ennek a komponensnek egyetlen belső állapota van. Ebben tárolom az éppen kiválasztott fül értékét. Ezt az állapotot a *Tabs* komponenst tudja megváltoztatni és azt a célt szolgálja, hogy eldöntsük melyik *TabPanel* jelenjen meg. A szaggatott vonal azt jelenti, hogy ezek a részfák nem mindig kerülnek megjelenítésre. Akkor látszódik a *TabPanel* tartalma, ha a komponens általunk megadott indexe és az állapotban tárolt érték megegyezik. Ellenkező esetben a részfát nem tartalmazza a DOM[[2]](#footnote-2).

Az *Exercise* komponens kompozícióját a következőképpen lehet illusztrálni:



3‑. ábra Exercise komponens felépítése

A 3‑6. ábra azért jeleníti meg egy szinten a *QuizForm* és *ProgrammingTask* komponenst, mivel egy szinten vannak meghivatkozva, az *Exercise* komponensben. Viszont a *QuizFrom* komponens gyerekeként fog megjelenni a DOM-ban.

## Kódfordítást végző szerver

### Architektúra

A kódfordítást egy NodeJS-en futó [express](https://www.npmjs.com/package/express) applikáció végzi. Az alkalmazás használ shell scripteket, amik tartalmaznak operációs rendszer specifikus parancsokat. Emiatt csak unix-szerű operációs rendszereken futtathatóak le. A környezetfüggetlenséget előtérbe helyezetve, ezért a szerver egy unix operációs rendszer alapú Docker konténerben fut. Pontosabban 3.12-es verziójú Alpine Linux-on. Az alkalmazás 14.15.0-as verziószámú NodeJS-t használ.

A Docker Engine alapvetően egy kliens-szerver alkalmazás. A klienst Docker CLI-nek hívjuk, a szervert pedig Docker daemon-nak az architektúrában. Ez a kettő egy REST API[[3]](#footnote-3)-n keresztül kommunikál. Ezt a 3‑7. ábra illusztrálja.



3‑. ábra Docker Engine architektúra

Az én szerverem is használja a Docker CLI által biztosított parancsokat, hogy kommunikálni tudjon a Docker daemon-al. Például konténerek létrehozásakor, amiben lefordulnak és futnak a felhasználó által küldött forráskódok. A [Docker CLI](https://pkgs.alpinelinux.org/package/edge/community/x86/docker-cli) már telepítve van a konténeremben, amikor elindítjuk azt. Ezt az image készítésekor már hozzáadom egy új rétegként, az operációs rendszer csomagkezelő rendszerének segítségével (APK[[4]](#footnote-4)). Az image készítéséhez a */backend/src/server* útvonalon található Dockerfile-t használom fel.

Mivel feltelepítettük a Docker CLI-t, így már képes értelmezni a docker szóval kezdődő parancsokat az operációs rendszer (pl.: docker ps), de hibába fogunk ütközni a parancsok futtatásakor, mivel nincs mögötte a szervert képező Docker daemon. Léteznek olyan kész Docker image-ek, amik tartalmazzák a szerver és a kliens részeket (pl.: [docker-in-docker](https://hub.docker.com/_/docker)), de ezekkel előfordulhatnak performancia problémák. A projekt fejlesztése során ezért a lent olvasható módszerrel oldottam meg a problémát.

A Docker úgy képes Windows alapon is Linux alapú konténereket futtatni, hogy maga a DockerDesktop is egy virtuális gépen fut.



‑. ábra Docker Desktop a Hyper-V Manager alkalmazásban

A Docker daemon háromfajta socketen keresztül tudja a kéréseket várni, unix, tcp, fd. Alapértelmezett esetben egy unix domain socket jön létre a */var/run/docker.sock* útvonalon (esetünkben ez a virtuális gépen belül). Eléréséhez *root* vagy a *docker* csoport jogosultságára van szükségünk (erről bővebben [itt](https://docs.docker.com/engine/reference/commandline/dockerd/) olvashatunk). Ezt a socketet felcsatolva egy konténerbe képesek vagyunk a host gépen található Docker daemon-nak parancsokat adni, abban az esetben, ha a konténer tartalmazza a Docker CLI-t.



‑. ábra Docker elérése konténeren belülről

A projektben ennek a socket-nek a felcsatolása a docker-compose fájlban található.



‑. ábra Docker socket felcsatolása compose fájl segítségével

### API

A szerverrel egy REST API-n keresztül lehet kommunikálni. Két darab végpont létezik:

Egy */ping*, ez a végpont csak egy pong üzenettel válaszol. Azt a célt szolgálja, hogy le tudjuk tesztelni, hogy a szerver elérhető-e.



‑. ábra Ping végpont

A logikát végző végpontot pedig a */api/compile* útvonalon érhetjük el. A HTTP metódusok közül, csak a POST-ot tudja értelmezni. A kérés *body* részébe egy JSON[[5]](#footnote-5)-t kell küldenünk, aminek két kulcsa lehet: code, args. A *code* mező kötelező, ha a Javascript értelmében falsy értéket küldünk 400-as hibaüzenetet fogunk válaszként kapni. Az *args* mezőben tudunk a programnak argumentumokat megadni. Ez egy szöveg típusú mező, szóközzel válasszuk el az argumentumokat. Válaszként szintén egy JSON-t kapunk, ami tartalmazza a program futásának eredményét és hibás futás esetén a hibaüzenetet.

Minden kéréskor, ha helyesen megadtuk a *code* mezőt. Elindítunk egy konténert, amiben futhat a küldött forráskódból készített program. A szerver egy scriptet futtat (*docker\_timeout.sh*), ami létrehozza a konténert, és gondoskodik annak eltávolításáról, illetve időkorlát túllépés esetén a megszakításról. Ezt úgy oldottam meg, hogy a konténert a háttérben indítom el, a –-detach vagy -d kapcsolóval. Majd a [timeout](https://man7.org/linux/man-pages/man1/timeout.1.html) és [docker wait](https://docs.docker.com/engine/reference/commandline/wait/) parancs segítségével addig blokkolom a script további részeinek lefutását, amíg a timeout-nak megadott időkorlát lejár, vagy a docker wait parancs végez. A docker wait úgy működik, hogy meg kell adni egy konténer azonosítót, és addig blokkolja a folyamatot, amíg a megadott azonosítójú konténer le nem áll. Ez jelezve a program lefutásának végét. A timeout pedig a megadott időkorlátig engedi futni az argumentumként megadott parancsot, ha ezt túllépi elküldi neki a kill szignált. A script további részében leállítom a konténert, lekérem a konténer log-ját és eltávolítom magát a konténert.



‑. ábra Compile végpont

A végpont logikáját leíró folyamatábrát a 3‑13. ábra illusztrálja.



**3‑13**. ábra A kódfordítás folyamata

A forráskód konténerbe juttatása:

Külső fájlokat többféleképpen tudunk a konténerbe juttatni. Az egyik ilyen módszer például, ha a Docker image buildelésekor bemásoljuk a fájlt a *COPY* utasítás segítségével. A Shell scriptet is így juttattuk be, ami a konténer belépési pontját képezi. A script fordítja-, és futtatja le a kapott forráskódot. Ezzel a megközelítéssel a következő probléma lép fel. Minden alkalommal újra kell építenünk a Docker image-et, ha megváltozik a fájl. A belépési pontot képző script esetén ez nem probléma, hiszen az nem változik. Ellenben, ha a kapott forráskódot akarnánk így a konténerbe juttatni, akkor minden új kérés esetén újra kéne buildelni az image-et, ami számítás-, és tárhelyköltséges.

A projekt során már használt Docker volume-okhoz fordulunk segítségül, hogy kikerüljük az image folytonos újraépítését. Lehetőségünk nyílik a Docker segítségével a fájlrendszerünk egy bizonyos részét felcsatolni a konténerbe. Ha a *host* gépen le tudnánk menteni a forrás fájlt és a konténer indításakor felcsatolni azt, akkor a problémánk meg is oldódna. Viszont mi magát a szervert is egy konténerbe futtatjuk, és ebből a konténerből indítjuk a további konténereket. Ezzel sajnos elveszítjük azt a lehetőséget, hogy *Host volume*-okat használjunk. A megoldást a *Named volume*-ok hozzák, amiket képesek vagyunk konténeren belülről is felcsatolni más konténerekre. Ezt a volume-ot a compose fájl segítségével hozzuk létre. A szerverre ezt a */usr/temp* útvonalra csatolom fel, és ide írjom ki fájlba azokat a forráskódokat, amit a kliensektől kapok. A szervernek ehhez a volume-hoz írási és olvasási joga is van. Ugyanezt a volume-ot csatolom fel a szerver által indított konténerre is, hogy ki tudja olvasni a forráskódot, és ezután lefordítsa és futtassa. A kliens konténer csak olvasási jogot kap, hogy esetleges kártékony kód esetén ne tudja elrontani más kliensek folyamatát.

3‑14. ábra Konténerek és volume-ok kapcsolata

### Build

A Docker image egy kétfázisú folyamat során készül el. Először átmásoljuk a forráskódot, és feltelepítjük a függőségeket. A *node\_modules* mappa bekerült a .dockerignore fájlba, hogy ez ne másolódjon át. Ennek az egyik oka, hogy lehet nem is létezik a könyvtár. A fontosabb indok pedig, hogy gyorsabb egyszerűen feltelepíteni újra a csomagokat. Ez főleg annak köszönhető, hogy nagyobb contextek esetén jelentősen megnő a számítási igény Docker részről. Az *npm ci* paranccsal telepítjük fel a csomagokat, mivel ez is jelentősen gyorsabb mint a sima *npm install*, viszont ehhez szükséges a *package-lock.json* fájl. Ezt ne töröljük ki. A csomagok telepítése után magát a részalkalmazást buildel-jük.

A NodeJS build folyamatot a webpack segítségével végezzük el. Alapvetően backend-es NodeJS alkalmazásokhoz, nem szoktak bundling tool-t használni, itt csak azért lett, hogy a szerver forráskódját le tudjuk minify-olni, hogy csökkenjen a fájlméret és így az image méret is. A build által generált fájlok a */backend/build/server* mappában találhatóak.

Miután ez sikeresen lefutott, a második fázisban készítem el magát a végleges image-et. Átmásolom a build által generált fájlokat, és azt a shell scriptet, amivel a fordító konténereket fogom futtatni. Futási jogot adok ennek a scriptnek, és végül beállítom a belépési pontot, hogy a konténer indításakor egyből elinduljon a szerver.

### Környezeti változók

* VOLUME\_NAME: a named volume-nak a neve, amiben a forráskódokat tároljuk ideiglenesen. Az alapértelmezett értéke: *diploma-temp-files*.
* COMPILE\_CONTAINER\_TIMEOUT: A konténer maximum élettartamát határozza meg. Egy lebegőpontos szám, opcionálisan a következő szuffixumok egyikével:
  + s: másodperc
  + m: perc
  + h: óra
  + d: nap

Alapértelmezett értéke: 30s

* COMPILE\_IMAGE\_NAME: Annak a Docker image-nek a neve, amelyből a fordító konténer példányok készülnek. Alapértelmezett értéke: *diploma-compile-client*

## Adatbázis

A content management system-ben létrehozott leckék és feladatok egy MySql adatbázis kezelő rendszer segítségével kerül fenntartásra. Ennek a verziószáma: 5.7. A MySql-nek a [hivatalos docker image](https://hub.docker.com/_/mysql)-ét használtam fel a projekthez. Alap esetben elveszítjük az adatbázis tartalmát a konténer eltávolításával, ha nem használunk volume-ot. Ezért hoztam létre a */var/lib/mysql* útvonalra egyet. Így a konténer élettartamtól függetlenné lehet tenni az adatbázis állapotát. A projekten named volume-ot használok file system-es mount volume helyett, ennek annyi oka volt, hogy így nem szemeteljük tele a git repository-t. Ezt ki lehetne kerülni gitignore fájllal is, de a biztonság kedvéért ezt használom, nehogy esetleges rossz git konfiguráció esetén bekerüljön a teljes adatbázis a verziókövető rendszerbe.

### Táblák



‑. ábra Tutorial tábla mezői

Mezők:

* Kötelező:
  + id (automatikusan kap értéket)
  + name
  + difficulty
  + url\_alias
  + exercise
* Opcionális:
  + status (alapértelmezett érték: draft)
  + slides
  + description



‑. ábra A home tábla mezői

Mezők:

* Kötelező:
  + content
  + contact\_email
  + contact\_phone
* Opcionális:
  + useful\_links



‑. ábra Az exercise tábla mezői

Mezők:

* Kötelező:
  + id (automatikusan értéket kap)
  + question
  + solution
  + name
* Opcionális:
  + description
  + quizzes



3‑18. ábra A táblák EK diagramja

### Környezeti változók

Az alábbi környezeti változók közül mind kötelező.

* MYSQL\_DATABASE (Az adatbázis neve)
* MYSQL\_USER (Adatbázis felhasználónév)
* MYSQL\_PASSWORD (Az adatbázis felhasználó jelszava)
* MYSQL\_ROOT\_PASSWORD (Az adatbázis root felhasználójának jelszava)

### Az adatbázis elérése

A következő kétféle módon tudjuk elérni a konténerben futó adatbáziskezelő alkalmazást.

1, Direktben parancssor segítségével

1. nyissunk meg egy parancssor ablakot, Windows esetén ez lehet az alap Windows parancssor vagy Powershell is.
2. Futtasuk a következő parancsot: *docker exec -it diploma\_mysql\_1 mysql -u directus –p* (itt a –u-t követő érték a MYSQL\_USER környezeti változónak megadott érték legyen). Írjuk be a jelszót (MYSQL\_PASSWORD környezeti változó értéke), és már tudjuk is használni az adatbáziskezelő programot.

Példa: 

2, A DockerDesktop alkalmazás segítségével

1. Az alkalmazásban kattintsunk a kis CLI ikonra a mysql konténernél. Erre a 3‑19. ábra tartalmaz példát.
2. A megjelenő terminál ablakban futtassuk a következő parancsot: *mysql –u <MYSQL\_USER> -p*
3. Írjuk be a MYSQL\_PASSWORD környezeti változónak megadott értéket, utána tudjuk használni az adatbáziskezelő alkalmazást

Példa:





3‑19. ábra Adatbáziskezelő program elérése

Ha jobban szeretjük a vizuálisan is látni a táblákat és annak tartalmait, az [alábbi cikkben](https://towardsdatascience.com/connect-to-mysql-running-in-docker-container-from-a-local-machine-6d996c574e55) arról olvashatunk, hogyan köthetünk össze egy ilyen GUI[[6]](#footnote-6) alkalmazást egy Dockerben futó MySQL adatbázissal.

## Tesztelés

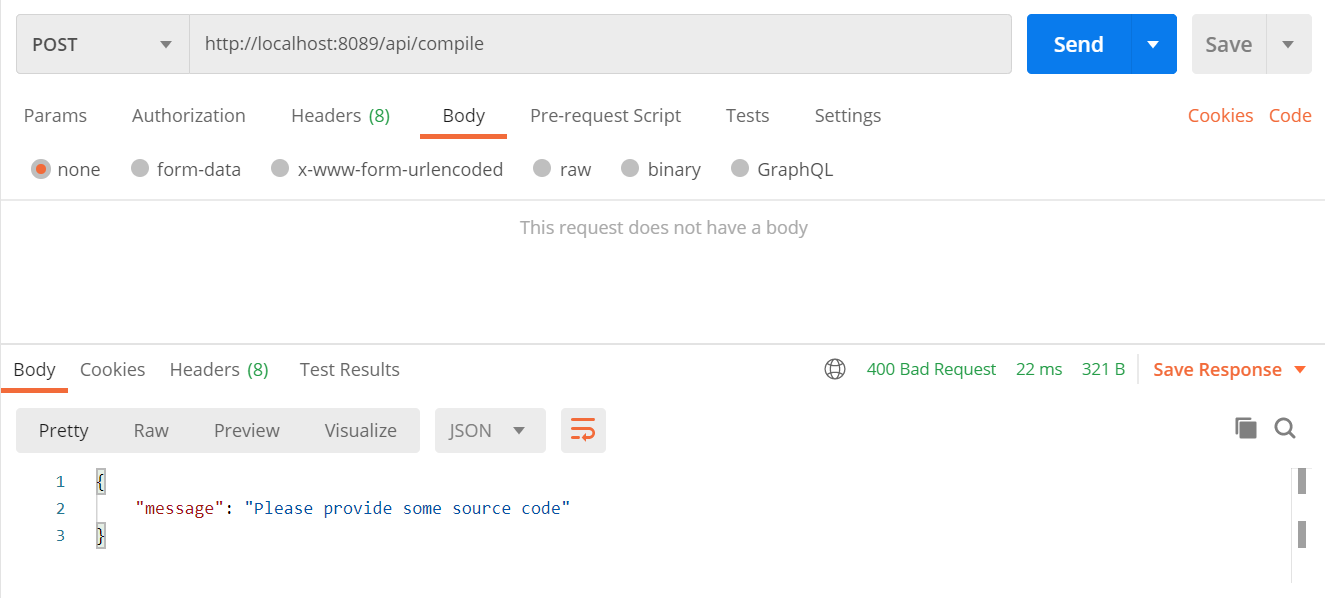
A kódfordítást végző szerveren manuális tesztet végeztem, aminek tesztesetei a [3.5.1-es fejezetben](#_Kódfordító_szerver_tesztelése) olvashatóak. A ReactJS alkalmazást unit teszteléssel is ellenőriztem. Mivel a CMS harmadik féltől származó alkalmazás, így azt nem teszteltem.

### Kódfordító szerver tesztelése

A tesztelést a Postman alkalmazás segítségével végeztem. A tesztgyűjtemény a */backend/test* mappa alatt található. A gyűjtemény kérései a következő teszteseteket fedik le:

Hiányzó forráskód:

A szerver 400-as státuszkódú hibaüzenettel válaszol.



Érvényes kérés:

A kérés jól lefut, a válasz body-ában látszódik az *stdout* és *stderr* mező.



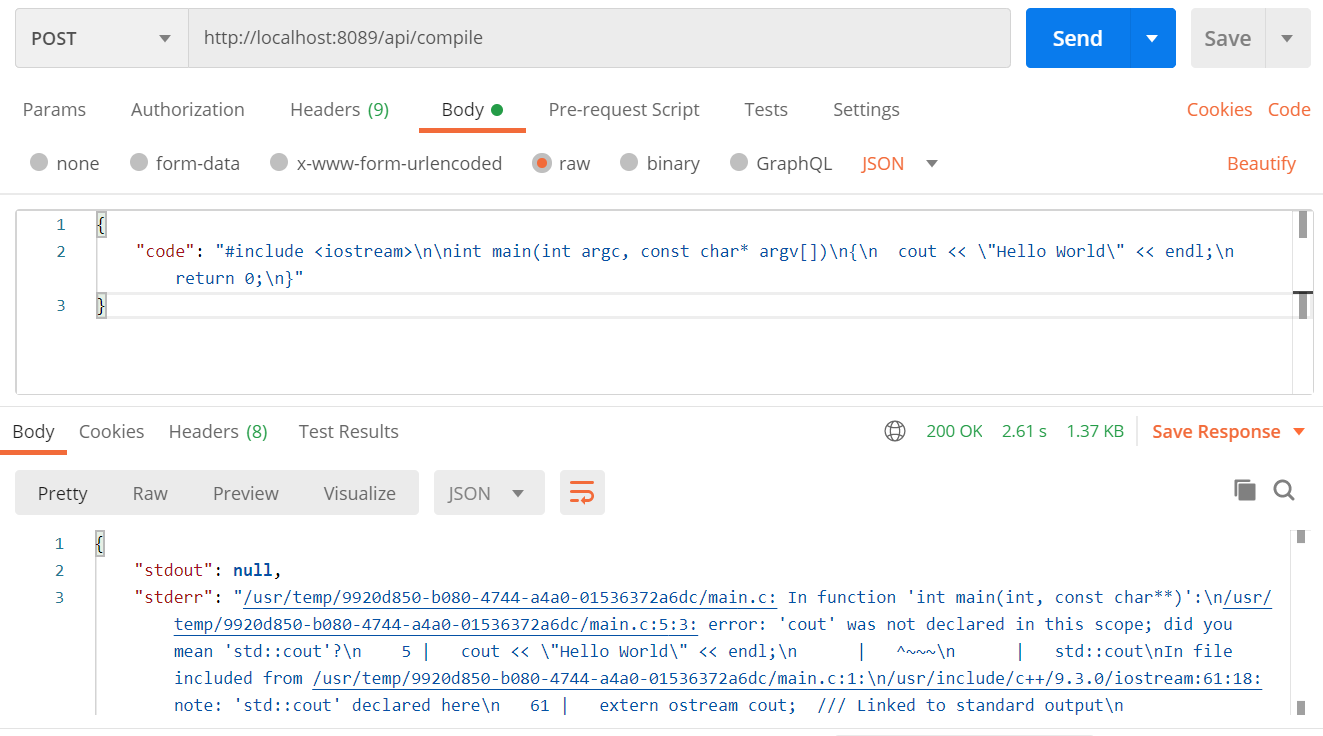
Érvényes kérés argumentumokkal:

A kérésben szereplő C++ forráskód kiíratja az argumentumokat, így azokat látnunk kell az *stdout* mezőben.



Érvényes kérés fordítási hibával:

A válasz *stderr* mezőjében látszik a fordítóprogram által generált hibaüzenet.



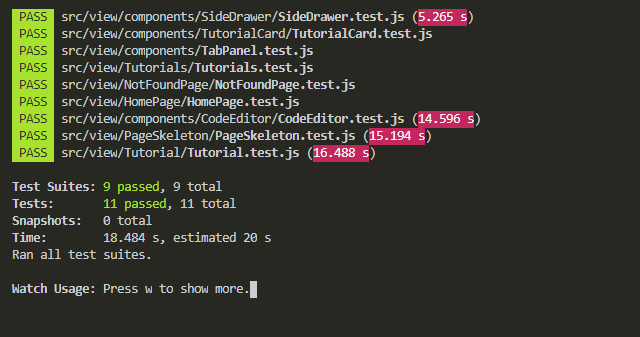
### ReactJS alkalmazás tesztelése

#### Unit tesztelés

Unit tesztek biztosítják, hogy a frontend kódbázis megfelelően működik. A tesztek írásához [Jest](https://jestjs.io/), és a [Tesing-library](https://testing-library.com/) tesztelési keretrendszert használtam.

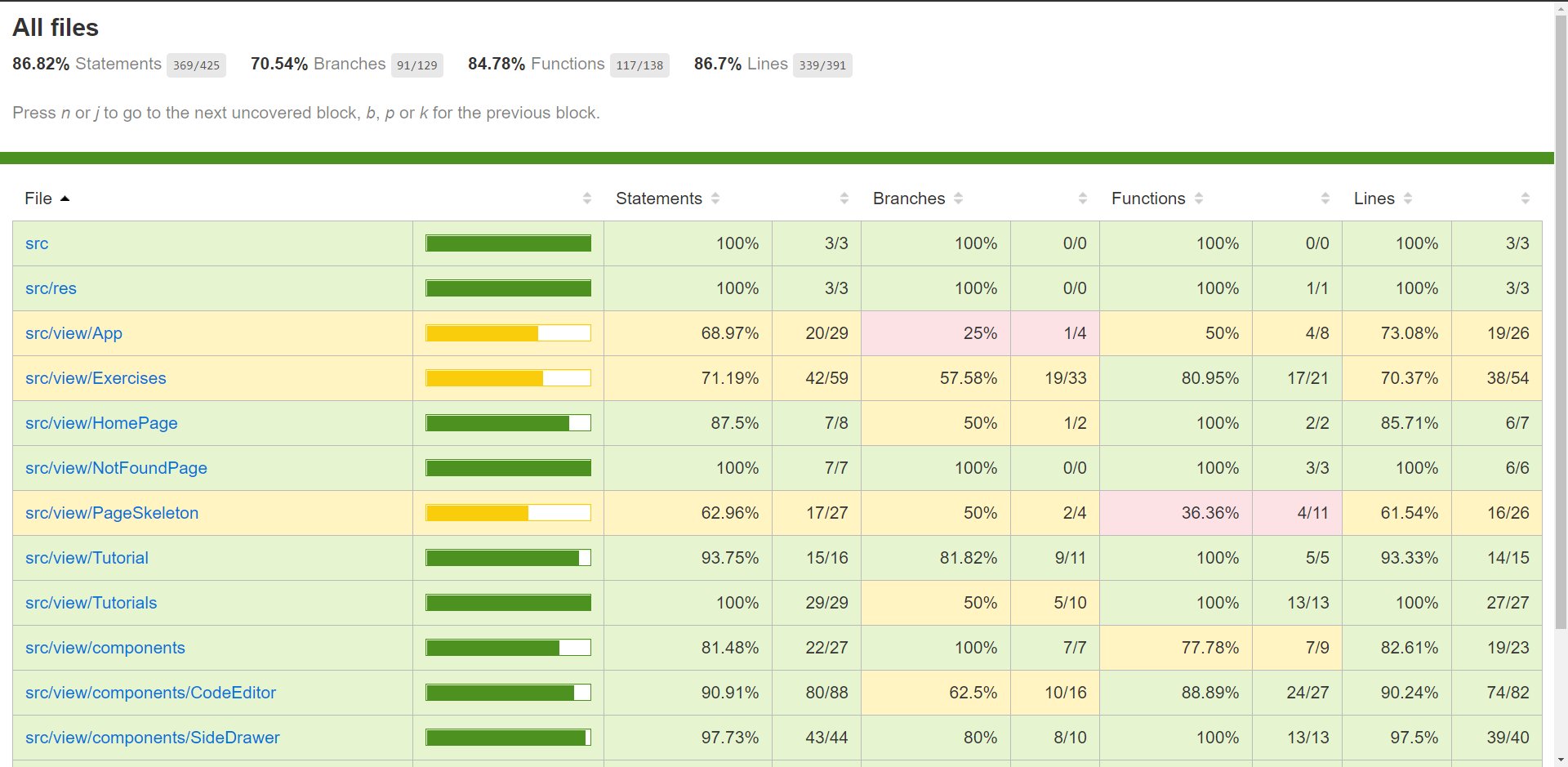
A Jest nyújtja a Javascript-hez szükséges teszt eszközöket, illetve modul mock-olást tesz lehetővé. Ez azért lényeges, mivel a tesztekben nem szeretnénk tényleges API hívásokat intézni a szerverek felé, és ezeket lehet ál hívásokkal helyettesíteni.

A Testing-library pedig a React specifikus eszközöket szolgáltatja. A React tesztelési keretrendszerek között a Testing-library-nek egy nagyobb vetélytársa van, az [Enzyme](https://enzymejs.github.io/enzyme/). Ez a két keretrendszer más nézőpontot képvisel a tesztelés terén. Az Enzyme inkább az implementációra helyezi a hangsúlyt, míg a Testing-library inkább a funkcionalitásra. Ezáltal alkalmas regressziós tesztelésre, amivel szélesebb használati eseteket fedhetünk le. A komponensekre a felhasználó-, nem pedig a fejlesztő szemszögéből kell tekinteni. A képernyőn szereplő elemeket úgy tudjuk elérni elsősorban, ahogy a felhasználó is tenné, a rajta szereplő szövegen keresztül. Például egy input mezőt annak címkéje vagy placeholder szövege alapján tudunk megkeresni. Egy screen reader-t használó felhasználó is hasonló élményben részesül az oldal használatakor. Ezáltal meggyőződhetünk, hogy az oldal széles felhasználóbázis számára elérhető. Mivel nem a komponensek implementációja van az előtérben, így egy esetleges kódbázis újradolgozás esetén a teszteket nem kell újraírni. Abban az esetben, ha a komponens működése nem változott, csak az implementációja. A projekthez ezért választottam a Testing-library-t. A tesztek a */frontend* könyvtárból az *npm test* paranccsal futtathatóak le.



‑. ábra A tesztek futási eredménye

Az [istanbul](https://istanbul.js.org/) eszköz által lehetőség nyílik megnézni, hogy a kódbázis mekkora részét van lefedve tesztekkel. Ezt a coverage report-ot az *npm run test:coverage* paranccsal lehet elkészíteni. Ez a script legenerálja a frontend mappába a coverage nevű könyvtárat. Ezen a mappán belül az *lcov-report/index.html* dokumentum megnyitásával tekinthető meg a jelentés. Ezt a jelentést különböző CI[[7]](#footnote-7) rendszerekbe is be lehet építeni. A 3‑21. ábrán látszik az összegző képernyő. Itt a teljes kódbázisra vonatkozó statisztika megjelenik. Innen navigálhatunk alkönyvtárakba vagy fájlokhoz.



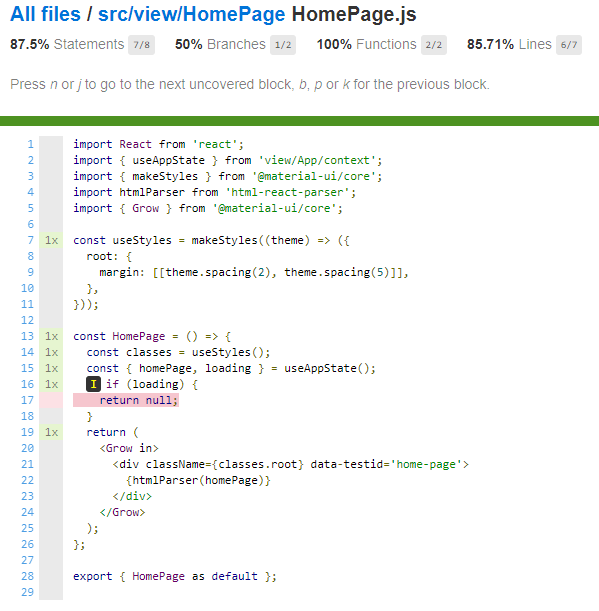
3‑. ábra Lefedettségi jelentés

Négy kategória van lefedettségi szempontból: statement, branch, function és line.

A statement és line kategóriát egybe is lehet venni. Ugye egy soron több statement is szerepelhet, de a kódformázó úgy van beállítva, hogy minden statement új sorban kezdődjön. A két érték között minimális eltérés lehet. A kódbázisban szereplő utasítások letesztelt arányát ábrázolja. A 3‑22. ábra által is bemutatott részletes nézeten, látni kis számmal a sorszám mellett, hogy az adott sort hányszor fedték le a tesztek.

A branch jelenti, hogy a tesztünk a lehetséges elágazások hány százalékát fedte le. Erre példa a 3‑22. ábra. Itt pirossal van kiemelve, melyik ágat nem fedi le a teszt. Itt az látszik, hogy a komponens nincs tesztelve, ha az alkalmazás még töltő állapotban lenne. Az alkalmazás akkor van töltő állapotban, amikor még függőben vannak az API kérések a CMS felé.

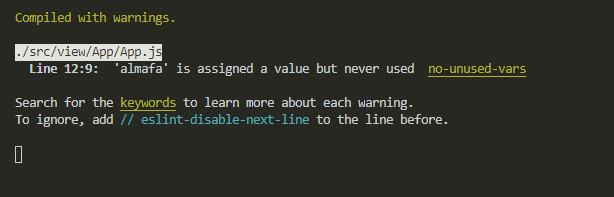
A function pedig szimplán a függvényeket nézi. A komponensek is függvényként vannak definiálva, így minden renderelésnél lefutnak. Ezért fordulhat elő, hogy egy nagyobb számot látunk a komponens definíciós sora mellett. Ez lényegében azt jelöli hányszor renderelődött újra a komponens a tesztek alatt.



3‑22. ábra Egy fájl lefedettségi jelentése

#### Statikus kódellenőrzés

Ezt az Eslint alkalmazás végzi, aminek megadhatóak szabályok, hogy milyen hibát okozható kódrészekre figyeljen. A fő konfigurációt a [React-app](https://www.npmjs.com/package/eslint-config-react-app)-hoz tartozó alapbeállítás adja. Ez olyan dolgokra figyel, hogy a HTML image tag-nek van-e helyettesítő szöveg megadva, vagy hogy helyes-e a HTML tag-ek egymásba ágyazása. A helyes HTML használaton kívűl a Javascript kódolási hibákat is kiemeli. Ilyen ha nem használt változó van deklarálva, vagy névelfedés van belső függvények között. Erre a 3‑23. ábra ad egy példát.



3‑. ábra Eslint hiba figyelmeztetés

Ez alapvetően nem okozna hibát a kód lefutásában, de hibaforrás lehet. Be lehet úgy is állítani a konfigurációt, hogy bizonyos szabályokra hibát dobjon figyelmeztetés helyett, megakadályozva a sikeres kódfordítást. Így kikényszerítve a figyelmeztetés javítását.

# Összefoglalás

A projekt elkészítése során számos új tudásra tettem szert, például a konténerizációra, amiről eddig csak felületes ismeretem volt. Szerveroldali Javascriptet is először a szakdolgozatomban használtam. Úgy gondolom, ezeknek még nagy hasznát fogom venni a karrierem későbbi időszakában. Az egyetemen szerzett széleskörű tudásnak köszönhetően gyorsan tudtam alkalmazkodni az új technológiákhoz, és segítettek a projekt architektúrájának megtervezésében. A használt keretrendszerek dokumentációjában elmerülve olyan ismeretekre is sikerült szert tennem, amiket jelenlegi munkahelyemen is tudtam használni, hogy javítsam a kódminőséget. Véleményem szerint jól sikerült elkülöníteni a teljes alkalmazást képző részmodulokat. Azok fejlesztése a többiektől elkülönítve is lehetséges. A legnagyobb kihívást a szerveroldali kódfordítás okozta. Ennek a résznek több verziója is létezett, de végül sikerült egy sokkal egyszerűbb és rövidebb megoldást találnom a korai megvalósításokhoz képest.

## Továbbfejlesztés lehetőségei

A dolgozat határideje előtt pár héttel kijött a Directus alkalmazásnak (ami a projektben a CMS modult képzi) egy újabb verziója. Hosszú évek óta ez volt a legnagyobb változtatás. Az új verzióban lehetőségünk nyílik MySQL-tól eltérő adatbáziskezelő program használatára, illetve nem muszáj Apache webszervert használni. Használható NodeJS alapú szervert is, hogy egységesebb legyen az architektúra a modulok között, és könnyebb legyen saját belső CMS modulok fejlesztése. Illetve a feladatok ellenőrzése jelenleg a frontenden történik. Ehhez viszont a megoldásokat is el kell küldenünk a frontend-nek, amit ügyesebb szemek észrevehetnek. Az ellenőrzés folyamatát át lehetne ültetni a kódfordító szerverre, egy új végpont bevezetésével.

1. Content Management System [↑](#footnote-ref-1)
2. Document object model [↑](#footnote-ref-2)
3. Representational State Transfer Application Programming Interface [↑](#footnote-ref-3)
4. Alpine Package Keeper [↑](#footnote-ref-4)
5. Javascript Object Notation [↑](#footnote-ref-5)
6. Graphical user interface [↑](#footnote-ref-6)
7. Continuous Integration [↑](#footnote-ref-7)